

CMSCONSTRUCT

IT Discovery Machine

Content Gathering Module

Find Endpoints

Chris Satterthwaite

Jan 22, 2018

Contents

Document Revisions.....	2
Introduction	3
Code Design	4
Flow Diagram	4
Pseudocode.....	4
Sample Output	7
Module Contents	13
Jobs	13
Find_ICMP_socket.....	13
Find_SNMP.....	13
Find_WMI.....	14
Find_SSH	15
Find_PowerShell	15
Scripts.....	16
User Guide	16
Loading protocol credentials	16
ITDM Job schedules	17
Troubleshooting.....	17
printDebug	17

Document Revisions

Author	Version	Date	Details
Chris Satterthwaite	1.0	1/22/2019	Document created

Introduction

This document details the 'findEndpoints' module written for IT Discovery Machine (ITDM).

Jobs in this module attempt to establish a connection to an endpoint on a specified protocol, by running through the pool of credentials available to the job. Those credentials are filtered by realm, client group, credential group, and potentially other job-level context. After establishing a connection, a job requests high level information about the “brain” and the “body”, as well as listed IP Addresses, before creating/relating the objects accordingly. The “brain” context will be stored in a Node object with the name and domain of the OS instance along with OS level details. The “body” context will be stored in a Hardware object with attributes corresponding to the hardware. Finally, the Protocol, Hardware, and IP Addresses are linked to the Node.

The level of information will vary across protocol types and endpoints. For example, SNMP hitting the MIB2 sys table will not have the same level of context as WMI's root/Win32 classes. Consequently, any protocol-based strategy for discovering remote endpoints should consider that the quality and context of related objects will vary based on protocol and configurations. Since shell based protocols like SSH or PowerShell on server endpoints are superior to SNMP, the endpoint queries should favor Shell over SNMP.

Generally speaking, you only need to use a single protocol per endpoint. So, if you assign an order to the jobs in this module protocol priority, in a synchronous way via job schedules, then you can have the automation select the best fit. But that's the broad stroke. Some jobs will require strategic overlap of protocol types for content gathering. For example, the server-side protocol of choice is a Shell. But if a company has Microsoft DNS and wants to pull DNS records via WMI instead of having to setup zone transfer rights and using a command line equivalent over shell. In such cases, you could simply tune the endpoint queries (i.e. job targets) to cover that scenario, or better yet, create a clone of the job that calls the same script but uses a custom endpoint query targeting one or more DNS servers.

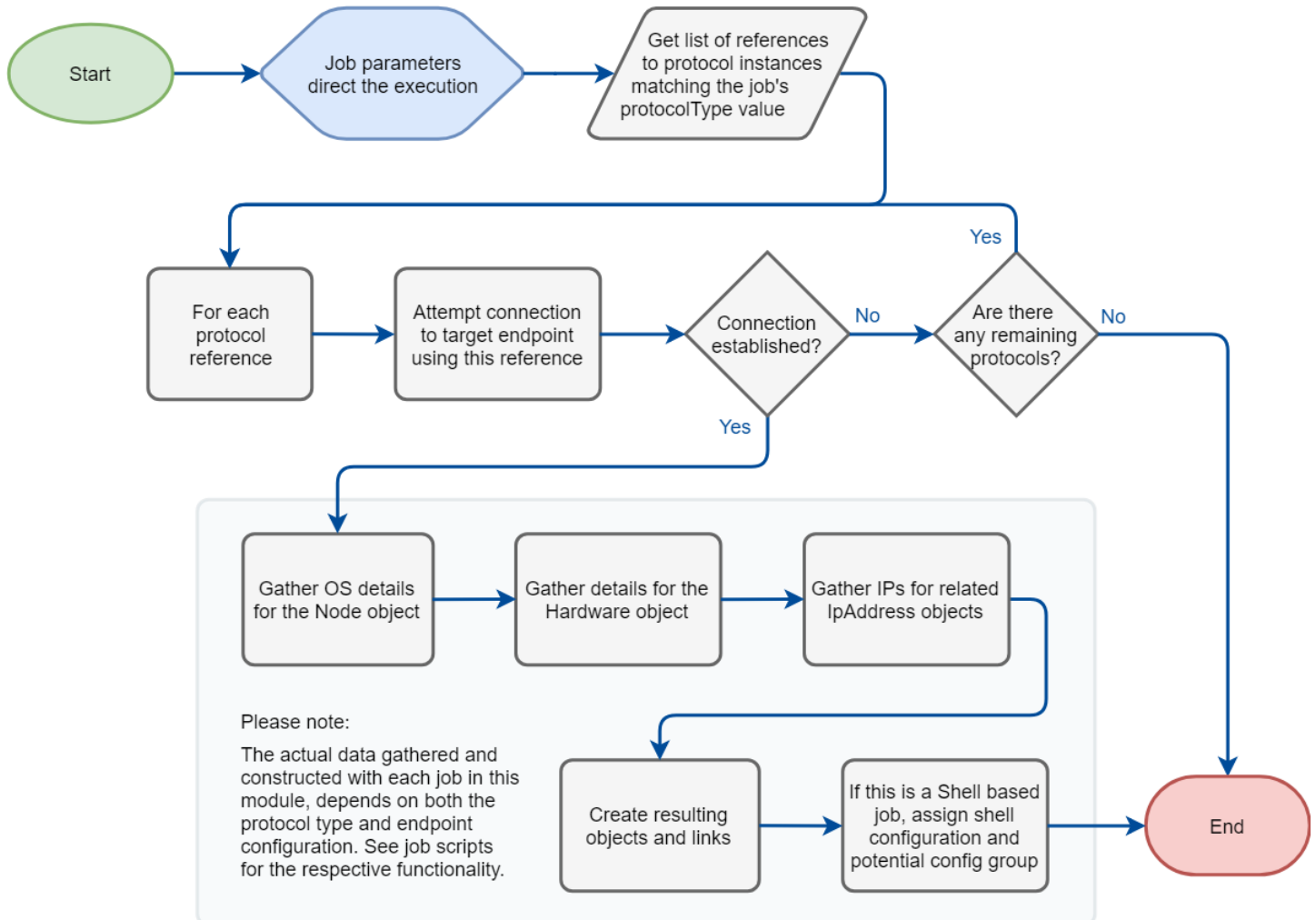
The flow of the 'findEndpoint' jobs are different than jobs that reference the already established protocol, since these jobs try different sets of credentials and determine failure according to their protocol and library. However, once a certain protocol has been established on an endpoint, all jobs that use said protocol can simply call the 'getClient' utility to connect to the previous context.

We leveraged established libraries for anything that was already openly available. Which meant leveraging a pure Python implementation for ICMP via raw sockets, fabric and paramiko for SSH, wmi from PyMI, pysnmp for SNMP. We wrote a library for PowerShell since there were no options.

Code Design

Flow Diagram

The following diagram shows the general flow for most of the find jobs, excluding ICMP:



Pseudocode

Pseudocode is an informal description of the design or algorithm of the code. This section takes the reader one level deeper into the working of the scripts. For specific functionality and code logic, please review the scripts themselves, which are written in Python with inline comments.

The '`find_ICMP_socket`' job accomplishes the following on all addresses within the boundary of the Realm defined by the job:

1. Attempt a raw socket connection to the IP (ping the IP)

2. If it passed, create an IP address object.
3. Update the ITDM job status and exit.

The '[find_WMI](#)' job accomplishes the following on previously created IPAddress objects that match the Realm defined by the job:

1. Go through WMI protocol entries and attempt a connection.
 - a. Note: this uses a compiled library call for security purposes, such that sensitive credential information is not visible to jobs.
 - b. Catch certain exceptions (RPC Errors, Access Errors) for better error message handling.
 - c. If the connection failed, continue to the next protocol entry.
2. If the connection was successful, run command to determine the OS variant of this endpoint.
3. Clear out errors from previous connection attempts (removing false negatives)
4. Query for the following:
 - a. OperatingSystem for OS level attributes
 - b. BIOS for serial number and firmware
 - c. ComputerSystem for name/domain
 - d. NetworkAdapterConfiguration for IPs
5. Update the job's runtime status to success
6. Create objects: Node, WMI, Hardware, IPAddress(es)
7. Exit the runtime

The '[find_SNMP](#)' job accomplishes the following on previously created IPAddress objects that match the Realm defined by the job:

1. Go through SNMP protocol entries and attempt to use the protocol:
[Note: this uses a compiled library call for security purposes, such that sensitive credential information is not visible to jobs.]
 - a. Query the base mib2 system table (1.3.6.1.2.1.1)
 - b. If the query failed, continue to the next protocol entry
2. If the query was successful, clear out errors from previous attempts (removing false negatives)
3. Query for the following:
 - a. Host device table (1.3.6.1.2.1.25.3.2.1.3)
 - b. IP tables (1.3.6.1.2.1.4.20.1.1)
4. Update the job's runtime status to success
5. Create objects: Node, SNMP, and IPAddress(es)
6. Exit the runtime

The '[find_SSH](#)' job accomplishes the following on previously created IPAddress objects that match the Realm defined by the job:

7. Go through SSH protocol entries and attempt a connection.

- a. Note: this uses a compiled library call for security purposes, such that sensitive credential information is not visible to jobs.
 - b. Catch certain exceptions (Connection Errors, Authentication Errors, Timeouts) for better error message handling.
 - c. If the connection failed, continue to the next protocol entry.
8. If the connection was successful, run command to determine the OS variant of this endpoint.
9. Clear out errors from previous connection attempts (removing false negatives)
10. Query for the following:
 - a. BIOS for serial number and firmware
 - b. name/domain
 - c. IP Addresses
11. Update the job's runtime status to success
12. Create objects: Node, SSH, Hardware, IPAddress(es)
13. Assign shell configuration and potential config group
14. Exit the runtime

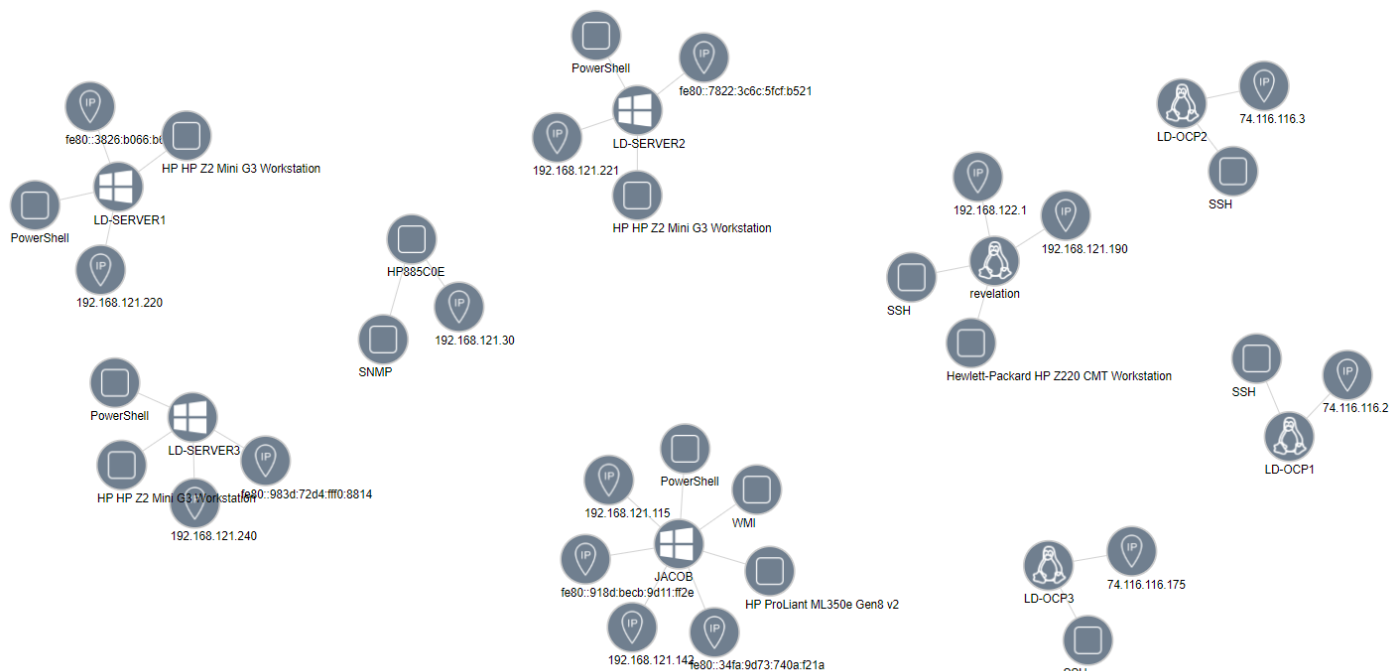
The 'find_PowerShell' job accomplishes the following on previously created IPAddress objects that match the Realm defined by the job:

1. Go through PowerShell protocol entries and attempt a connection.
 - a. Note: this uses a compiled library call for security purposes, such that sensitive credential information is not visible to jobs.
 - b. Catch certain exceptions (Connection Errors) for better error message handling.
 - c. If the connection failed, continue to the next protocol entry.
2. If the connection was successful, run command to determine the OS variant of this endpoint.
3. Clear out errors from previous connection attempts (removing false negatives)
4. Query for the following:
 - a. OperatingSystem for OS level attributes
 - b. BIOS for serial number and firmware
 - c. ComputerSystem for name/domain
 - d. NetworkAdapterConfiguration for IPs
5. Update the job's runtime status to success
6. Create objects: Node, PowerShell, Hardware, IPAddress(es)
7. Assign shell configuration and potential config group
8. Exit the runtime

Sample Output

Let's start with a graphical view of the samples below. This is created through an external visualizer and helps illustrate the data created by the jobs in this module.

Below we see three Windows endpoints (LD-SERVER1, LD-SERVER2, and LD-SERVER3) were established through the PowerShell job. The fourth Windows endpoint (JACOB) ran both WMI and PowerShell jobs. The Linux endpoints (LD-OCP1, LD-OCP2, LD-OCP3, revelation) were established through the SSH job. Notice that the SSH job did not find and connect hardware for the three OCP machines (which are running on virtual hardware), but it did for the one running on physical hardware (revelation). And last is the HP printer (HP885C0E) established through the SNMP job.



Now let's go through some details on the protocols above. These are obtained through REST API calls; we are showing the Insomnia utility below.

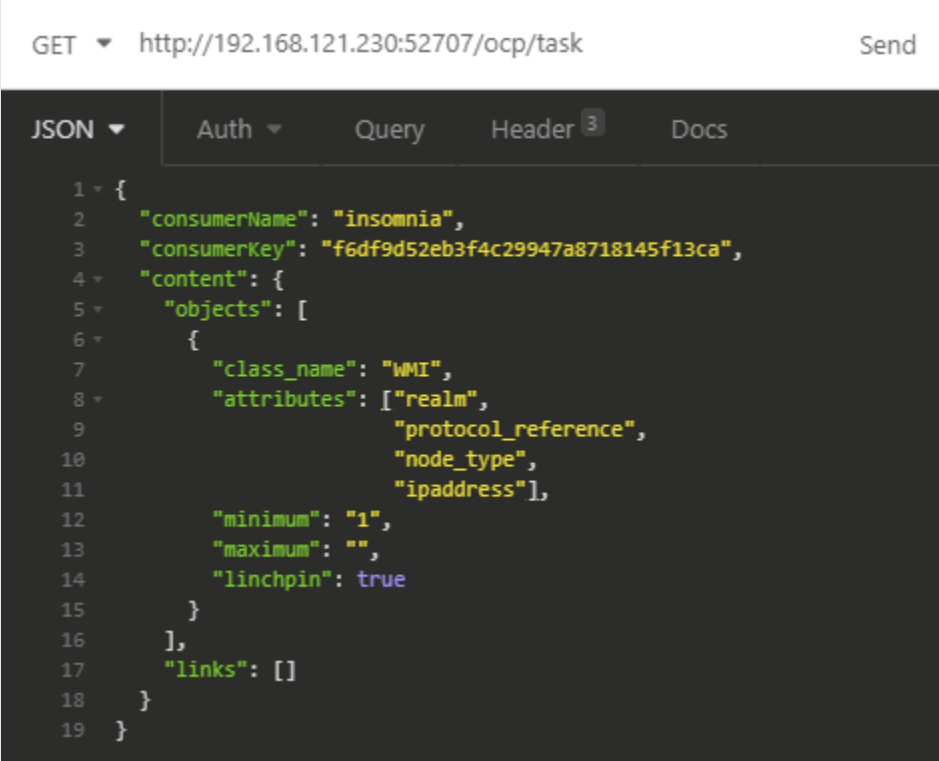
First we ask for created SNMP instances with the following query:

```
GET http://192.168.121.230:52707/ocp/task Send
JSON Auth Query Header 3 Docs
1 {
2   "consumerName": "insomnia",
3   "consumerKey": "f6df9d52eb3f4c29947a8718145f13ca",
4   "content": {
5     "objects": [
6       {
7         "class_name": "SNMP",
8         "attributes": ["realm",
9                       "protocol_reference",
10                      "node_type",
11                      "ipaddress"],
12         "minimum": "1",
13         "maximum": "",
14         "linchpin": true
15       }
16     ],
17     "links": []
18   }
19 }
```

Sample SNMP instance follows, which is the protocol instance for the HP printer shown above:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "43",
    "node_type": "Unknown",
    "ipaddress": "192.168.121.30"
  },
  "class_name": "SNMP",
  "identifier": "b6bdad8eeea44679bc3cccd4c5bb3fef",
  "label": "SNMP"
}
```


Now we will ask for WMI instances with this query:



```
GET http://192.168.121.230:52707/ocp/task Send

JSON Auth Query Header 3 Docs

1 {
2   "consumerName": "insomnia",
3   "consumerKey": "f6df9d52eb3f4c29947a8718145f13ca",
4   "content": {
5     "objects": [
6       {
7         "class_name": "WMI",
8         "attributes": ["realm",
9                       "protocol_reference",
10                      "node_type",
11                      "ipaddress"],
12         "minimum": "1",
13         "maximum": "",
14         "linchpin": true
15       }
16     ],
17     "links": []
18   }
19 }
```

Sample WMI instance, which is the protocol instance on JACOB from the initial snapshot:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "39",
    "node_type": "Windows",
    "ipaddress": "192.168.121.115"
  },
  "class_name": "WMI",
  "identifier": "456c4ee76ecb49d49fa0059e84d7fb55",
  "label": "WMI"
}
```

Both the SNMP and WMI instances are similar in that they have realm, protocol_reference, node_type, and ipaddress. The shell objects (SSH and PowerShell) have a bit more context.

We can ask for SSH instances apart from PowerShell instance, as we did above with SNMP and WMI. But since both shell types inherent from a shared Shell, we can request both at once with this query:

```
GET http://192.168.121.230:52707/ocp/task Send
JSON Auth Query Header 3 Docs
1 {
2   "consumerName": "insomnia",
3   "consumerKey": "f6df9d52eb3f4c29947a8718145f13ca",
4   "content": {
5     "objects": [
6       {
7         "label": "SHELL",
8         "class_name": "Shell",
9         "attributes": ["realm",
10                       "protocol_reference",
11                       "ipaddress",
12                       "parameters"],
13         "minimum": "1",
14         "maximum": "",
15         "linchpin": true
16       }
17     ],
18     "links": []
19   }
20 }
```

The following five snapshots show different types of results; take special notice of the parameters and the configGroup, which direct command execution in shell based scripts.

The first one matches a Windows Server 2012 R2 config group:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "35",
    "ipaddress": "192.168.121.115",
    "parameters": {
      "osType": "Windows",
      "configGroup": "Microsoft_Corporation_HP_Microsoft_Windows_Server_2012_R2_Standard_6.3.9600",
      "commandsToTransform": {
        "netstat": "netstat",
        "ping": "ping"
      }
    }
  },
  "class_name": "PowerShell",
  "identifier": "1f55030758384471a9553019e7020a33",
  "label": "SHELL"
},
```

The next one matches a Windows Workstation 10 Pro config group:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "34",
    "ipaddress": "192.168.121.221",
    "parameters": {
      "osType": "Windows",
      "configGroup": "HP_Microsoft_Windows_10_Pro_10.0.17134",
      "commandsToTransform": {
        "netstat": "netstat",
        "python": "python",
        "ping": "ping"
      }
    }
  },
  "class_name": "PowerShell",
  "identifier": "a76e520d97cf4bc38c403ac96539af86",
  "label": "SHELL"
},
```

This one shows a match on a CentOS 7 Core config group:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "36",
    "ipaddress": "74.116.116.2",
    "parameters": {
      "osType": "Linux",
      "configGroup": "__CentOS_Linux_7_(Core)",
      "commandsToTransform": {
        "sudo": "sudo",
        "lsof": "lsof",
        "ss": "ss",
        "rpm": "/bin/rpm",
        "ls": "ls --color=never",
        "perl": "perl"
      }
    }
  },
  "class_name": "SSH",
  "identifier": "22cbe42408aa4119af052197bd8f2207",
  "label": "SHELL"
},
```

This Linux endpoint did not match any config groups, and was dropped into the “default” group:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "37",
    "ipaddress": "74.116.116.175",
    "parameters": {
      "osType": "Linux",
      "configGroup": "default",
      "commandsToTransform": {
        "sudo": "sudo",
        "lsof": "lsof",
        "ss": "ss",
        "rpm": "/bin/rpm",
        "ls": "ls --color=never",
        "perl": "perl"
      }
    }
  },
  "class_name": "SSH",
  "identifier": "6c909328205c46899acfbf41c7a6c267",
  "label": "SHELL"
},
```

And this Linux endpoint matched a Red Hat EL 7.5 config group. Notice that the commandsToTransform on this endpoint are different from the two previous Linux OS endpoints, which we will discuss later:

```
{
  "data": {
    "realm": "default",
    "protocol_reference": "33",
    "ipaddress": "192.168.121.190",
    "parameters": {
      "osType": "Linux",
      "configGroup": "_Hewlett-Packard_Red_Hat_Enterprise_Linux_Workstation_7.5_(Maipo)",
      "commandsToTransform": {
        "sudo": "sudo",
        "netstat": "netstat",
        "lsof": "/usr/sbin/lsof",
        "ss": "/usr/sbin/ss",
        "rpm": "/bin/rpm",
        "ls": "ls --color=never",
        "perl": "perl"
      }
    }
  },
  "class_name": "SSH",
  "identifier": "89208be6bd754340a4fb343903e0fc41",
  "label": "SHELL"
},
```

Module Contents

Jobs

For general information on jobs (e.g. standard sections, parameter descriptions, general usage), refer to the Job Descriptor document.

Initially there are five (5) jobs available in this module. You may add more for additional protocol coverage, and you may later clone/update some for targeted endpoint discovery. But there should be at least one job defined per protocol.

File name	Description
find_ICMP_socket.json	Attempt a raw socket connection to endpoints (i.e. ping IPs)
find_SNMP.json	Attempt SNMP connection to endpoint; if successful, create Node, SNMP, and IP objects.
find_WMI.json	Attempt WMI connection to endpoint; if successful, create Node, WMI, Hardware, and IP objects.
find_SSH.json	Attempt SSH connection to endpoint; if successful, create Node, SSH, Hardware, and IP objects.
find_PowerShell.json	Attempt PowerShell connection to endpoint; if successful, create Node, PowerShell, Hardware, and IP objects.

There isn't much difference across the job definitions, but they are listed for completeness.

Find_ICMP_socket

Main meta-data section:

```
"jobName" : "find_ICMP_socket_weekly",
"realm" : "default",
"clientGroup" : "default",
"credentialGroup" : "default",
"protocolType" : "ProtocolIcmp",
"numberOfJobThreads" : 100,
"jobScript" : "find_ICMP_socket",
"endpointScript" : "realmScope",
"endpointIdColumn" : "value",
```

Comments:

- This job is configured to use 100 parallel threads on the clients.
- The entry script is "find_ICMP_socket.py", found in the module's script directory.
- The endpoint script is "realmScope.py", found in the module's endpoint directory.

Find_SNMP

Main meta-data section:

```

"jobName" : "find_SNMP",
"realm" : "default",
"clientGroup" : "default",
"credentialGroup" : "default",
"protocolType" : "ProtocolSnmp",
"numberOfJobThreads" : 30,
"jobScript" : "find_SNMP",
"endpointScript" : "ipsCreatedInRealm",
"endpointIdColumn" : "value",

```

Comments:

- This job is configured to use 30 parallel threads on the clients.
- The job directs the platform (via protocolType) to prepare credentials for SNMP.
- The entry script is “find_SNMP.py”, found in the module’s script directory.
- The endpoint script is “ipsCreatedInRealm.py”, found in the module’s endpoint directory.

Input parameters section:

```

"inputParameters" : {
    "queryForFQDN" : true,
    "printDebug" : false
}

```

Parameter descriptions:

Parameter name	Description
queryForFQDN	This enables a secondary lookup for the FQDN. If a domain was not found with the sys name, then this setting directs the job to attempt to get it from the socket library’s getfqdn(ip) method.
printDebug	This enables logging specific to a job, instead of working off global log levels; when true, all runtime.logger.report() lines are printed.

Find_WMI

Main meta-data section:

```

"jobName" : "find_WMI",
"realm" : "default",
"clientGroup" : "default",
"credentialGroup" : "default",
"protocolType" : "ProtocolWmi",
"numberOfJobThreads" : 30,
"jobScript" : "find_WMI",
"endpointScript" : "ipsCreatedInRealm",
"endpointIdColumn" : "value",

```

Comments:

- This job is configured to use 30 parallel threads on the clients.
- The job directs the platform (via protocolType) to prepare credentials for WMI.
- The entry script is “find_WMI.py”, found in the module’s script directory.
- The endpoint script is “ipsCreatedInRealm.py”, found in the module’s endpoint directory.

Find_SSH

Main meta-data section:

```
"jobName" : "find_SSH",  
"realm" : "default",  
"clientGroup" : "default",  
"credentialGroup" : "default",  
"protocolType" : "ProtocolSsh",  
"numberOfJobThreads" : 30,  
"jobScript" : "find_SSH",  
"endpointScript" : "ipsCreatedInRealm",  
"endpointIdColumn" : "value",  
"loadConfigGroups" : true,
```

Comments:

- This job is configured to use 30 parallel threads on the clients.
- The job directs the platform (via protocolType) to prepare credentials for SSH.
- The job directs the platform to (via loadConfigGroups) to load the config groups.
- The entry script is “find_SSH.py”, found in the module’s script directory.
- The endpoint script is “ipsCreatedInRealm.py”, found in the module’s endpoint directory.

Find_PowerShell

Main meta-data section:

```
"jobName" : "find_PowerShell",  
"realm" : "default",  
"clientGroup" : "default",  
"credentialGroup" : "default",  
"protocolType" : "ProtocolPowerShell",  
"numberOfJobThreads" : 30,  
"jobScript" : "find_PowerShell",  
"endpointScript" : "ipsCreatedInRealm",  
"endpointIdColumn" : "value",  
"loadConfigGroups" : true,
```

Comments:

- This job is configured to use 30 parallel threads on the clients.
- The job directs the platform (via protocolType) to prepare credentials for PowerShell.
- The job directs the platform to (via loadConfigGroups) to load the config groups.
- The entry script is “find_PowerShell.py”, found in the module’s script directory.
- The endpoint script is “ipsCreatedInRealm.py”, found in the module’s endpoint directory.

Scripts

There are several scripts available in this module and the number will increase as we increase the number of available protocols and tested endpoints:

File name	Description
find_ICMP_socket.py	Called by the ICMP job
find_SNMP.py	Called by the SNMP job
find_WMI.py	Called by the WMI job
find_SSH.py	Called by the SSH job
find_SSH_Linux.py	Library; called by find_SSH.py when the OS is Linux
find_SSH_Apple.py	Stubbed library; needs tested on customers with this OS type
find_SSH_AIX.py	Stubbed library; needs tested on customers with this OS type
find_SSH_HPUX.py	Stubbed library; needs tested on customers with this OS type
find_SSH_Solaris.py	Stubbed library; needs tested on customers with this OS type
find_PowerShell.py	Called by the PowerShell job
find_PowerShell_Windows.py	Library; called by find_PowerShell.py when the OS is Windows
find_PowerShell_Apple.py	Stubbed library; needs tested on customers running PowerShell Core (open source version) with this OS type
find_PowerShell_Linux.py	Stubbed library for future PowerShell Core support

User Guide

Follow the steps in this section before enabling jobs for the first time.

Loading protocol credentials

Work with security to gather username/password credentials to use when running these jobs. After getting those, you need to provide them to ITDM. The command line utility for this is found in the framework/lib sub-directory, and is called “createProtocolEntryUtil”.

Sample output follows from an execution of the utility:

```
PS D:\Software\CMS Construct\ITDM\server\framework\lib> python .\createProtocolEntryUtil.py
```


Protocol Type (sample entries: ['ProtocolSnmp', 'ProtocolWmi', 'ProtocolSsh', 'ProtocolPowerShell']):
[ProtocolSsh](#)
Realm (valid entries: ['default', 'customDomain']): [default](#)
credential_group: [default](#)
active: [1](#)
port:
description: [Linux ITDM account](#)
user: [itdm](#)
password:
retype password:

Note: typing something meaningful in the “description” may help future maintenance.

ITDM Job schedules

Open the jobs defined in this module, and set the scheduler accordingly:

```
./content/contentGathering/findEndpoints/job/*.json
```

Make sure to keep routine maintenance windows in mind while scheduling.

The protocol-based 'find' jobs are set to run once every two weeks by default, because they create objects that should not often change (Node, Hardware, Protocol, and IP Addresses). After running the first time, you want to ensure it runs after passwords change.

For customers that provision servers regularly during the week, you can create a copy of the jobs and give it a new endpoint query or endpoint script, specifically looking for new IPs. That way the copied version can run daily to find new servers, but the scope is limited to only new IPs instead of running across the full Realm. You can seed IPs from the ticketing system or an IPAM solution if you have those options available. Alternatively, you can run the ICMP job daily and use the ping sweep to find new IPs, which become targeted endpoints for your daily find job.

Troubleshooting

printDebug

The first step in troubleshooting is to enable the printDebug input parameter.

The printDebug parameter controls job-specific logging. When set to **true**, all runtime.logger.report() lines in job scripts are printed to the logs. Leaving this on all the time will consume unnecessary system resources, but enabling it is an obvious first step to troubleshooting:

```
"inputParameters" : {  
    "printDebug" : true  
}
```